Visualizing Call Graphs

Thomas D. LaToza & Brad A. Myers,
VL/HCC 2011

Summary by Prof. Thomas LaToza
SWE 795, Spring 2017
Software Engineering Environments

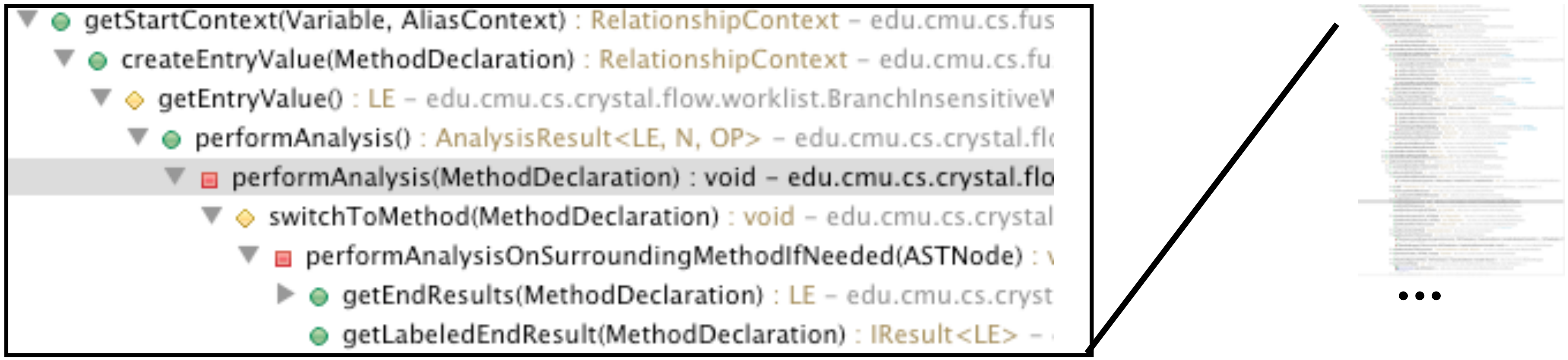
Motivation: Understanding control flow is hard

- Answering reachability questions frequent challenge in debugging & investigating implications of code

error prone	caused **50%** of bugs
frequent	>**9** times a day
hard	**82%** agree
time consuming	**tens** of minutes to answer

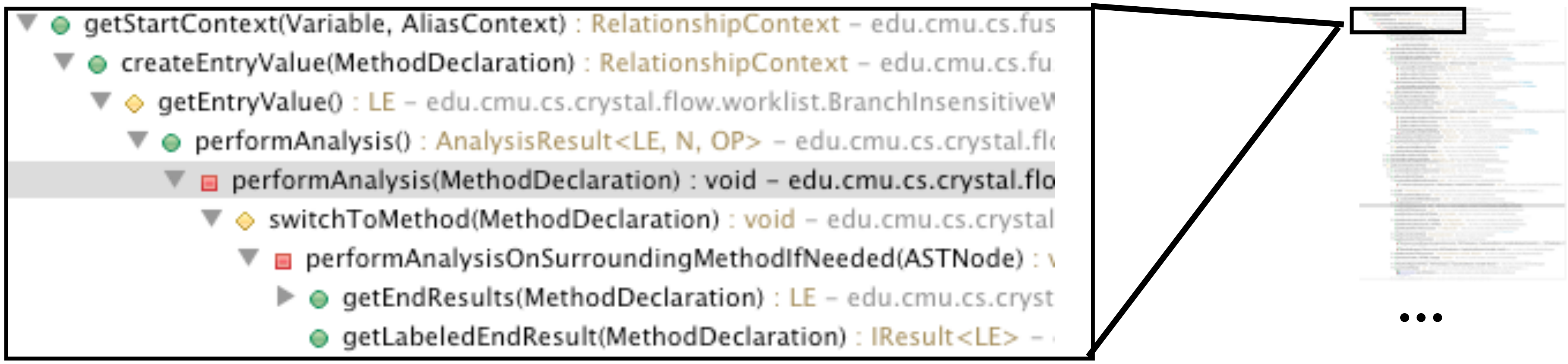
not easier or less frequent with knowledge or expertise

- Underlying cause: making foraging decisions across calls



```
▼ ● getStartContext(Variable, AliasContext) : RelationshipContext – edu.cmu.cs.fus
  ▼ ● createEntryValue(MethodDeclaration) : RelationshipContext – edu.cmu.cs.fu
    ▼ ◇ getEntryValue() : LE – edu.cmu.cs.crystal.flow.worklist.BranchInsensitiveW
      ▼ ● performAnalysis() : AnalysisResult<LE, N, OP> – edu.cmu.cs.crystal.fl
        ▼ ■ performAnalysis(MethodDeclaration) : void – edu.cmu.cs.crystal.flo
          ▼ ◇ switchToMethod(MethodDeclaration) : void – edu.cmu.cs.crystal
            ▼ ■ performAnalysisOnSurroundingMethodIfNeeded(ASTNode) : v
              ▶ ● getEndResults(MethodDeclaration) : LE – edu.cmu.cs.cryst
                ● getLabeledEndResult(MethodDeclaration) : IResult<LE> –
```

Searching along call graphs



```
getStartContext(Variable, AliasContext) : RelationshipContext – edu.cmu.cs.fus
  createEntryValue(MethodDeclaration) : RelationshipContext – edu.cmu.cs.fu
    getEntryValue() : LE – edu.cmu.cs.crystal.flow.worklist.BranchInsensitiveW
      performAnalysis() : AnalysisResult<LE, N, OP> – edu.cmu.cs.crystal.fl
        performAnalysis(MethodDeclaration) : void – edu.cmu.cs.crystal.flo
          switchToMethod(MethodDeclaration) : void – edu.cmu.cs.crystal
            performAnalysisOnSurroundingMethodIfNeeded(ASTNode) : v
              getEndResults(MethodDeclaration) : LE – edu.cmu.cs.cryst
                getLabeledEndResult(MethodDeclaration) : IResult<LE> –
```

Many methods, some of them are task relevant

- Finding them is hard...
- Information foraging models whole debugging / investigation task as traversing relationships to find search targets (prey) [Lawrance+2011]

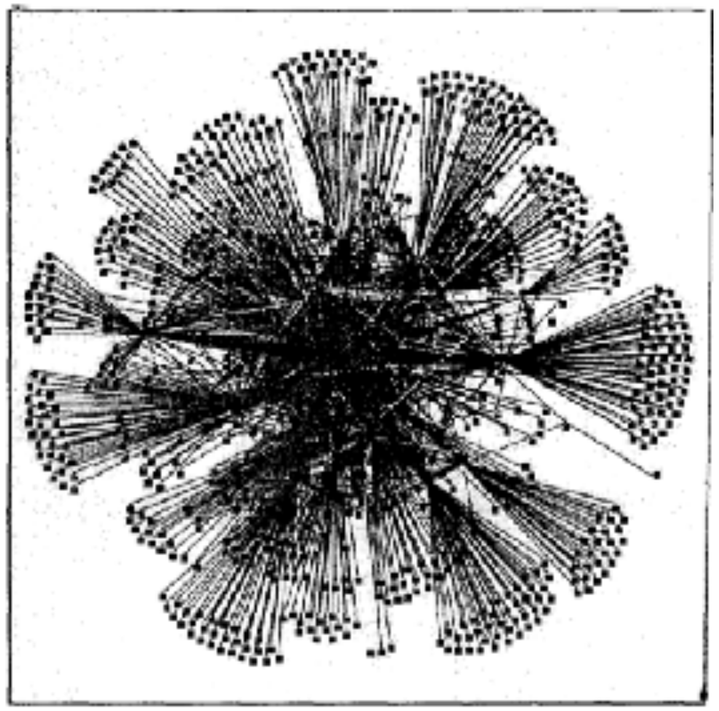
But developers search for statements by **attribute** (e.g., field writes) and **partial** name.

Design requirements for code exploration

Finding	**Implication**
search for statements by **attribute** (e.g., field writes) and **partial** name.	Configurable search dialog, incrementally match statements
rapidly investigate, never returning to most methods.	Expandable details on demand, browser style history navigation
explore **huge** call graphs, but task relevant portion small.	Only show the (task relevant) methods developers select.
reason about causality, class membership, ordering, choice, repetition.	Overview this information in visualization of callgraph
get **lost** and disoriented reading through code in disparate places.	Link callgraph to editor to navigate code.

Existing tools don't solve the problem

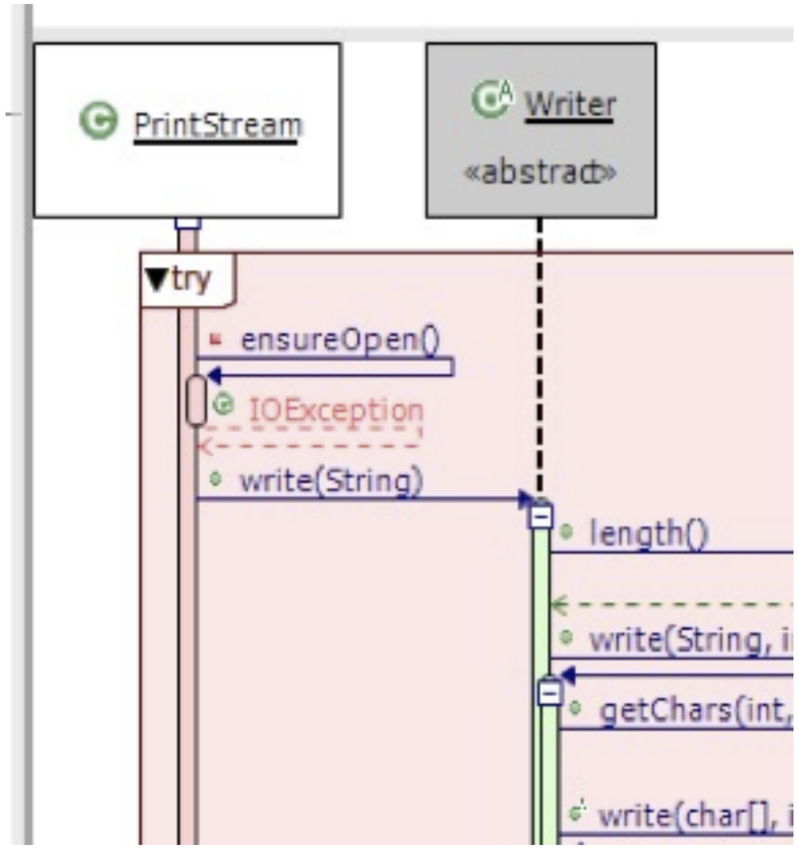
Graph visualizations



SHriMP [Storey+95]

-not task specific
-no search
-no ordering, class membership....

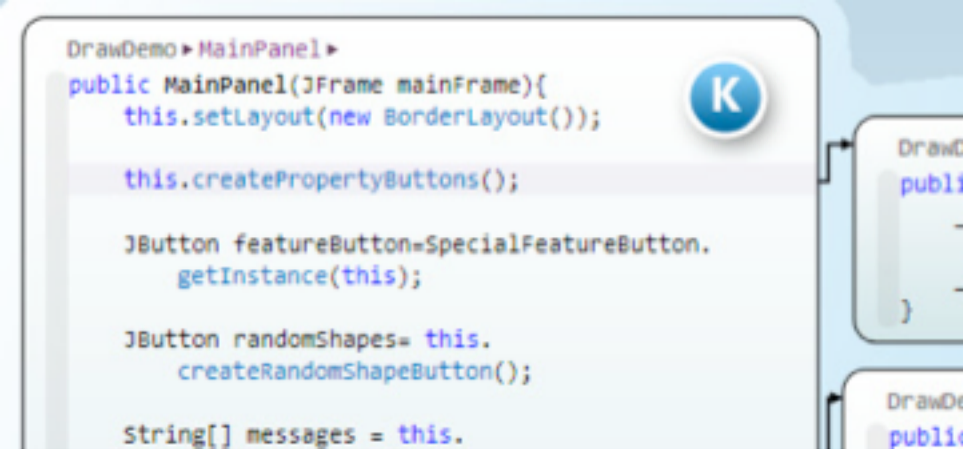
UML Sequence Diagrams



Diver [Bennet+07]

-not task specific
-not compact

Maps of code



```
DrawDemo ▸ MainPanel ▸
public MainPanel(JFrame mainFrame){
    this.setLayout(new BorderLayout());

    this.createPropertyButtons();

    JButton featureButton=SpecialFeatureButton.
        getInstance(this);

    JButton randomShapes= this.
        createRandomShapeButton();

    String[] messages = this.
```

Code bubbles [Bragdon+10]

-can't search over paths
-don't compactly encode ordering, repetition, conditionals, ...

Reacher

Designed a tool for understanding, exploring, and reasoning about call graphs

Implemented as an Eclipse plugin for Java

- Generates static call graphs with fast feasible path analysis
- Visualization built on Prefuse visualization toolkit [Heer+05]

Helps to

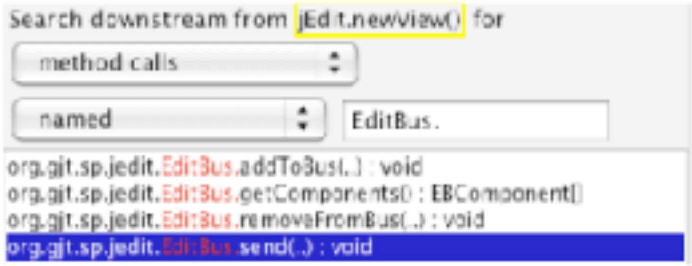
find statements

understand call graphs

stay oriented

by

entering searches



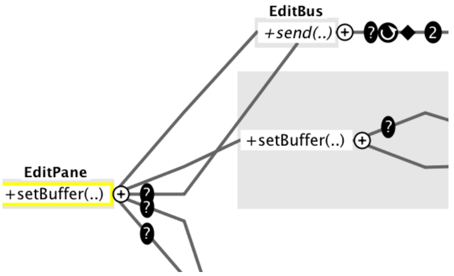
Search downstream from jEdit.newView() for

method calls

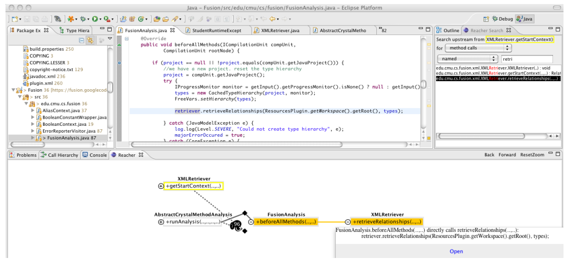
named EditBus.

org.gjt.sp.jedit.EditBus.addToBus(..) : void
org.gjt.sp.jedit.EditBus.getComponents() : EBComponent[]
org.gjt.sp.jedit.EditBus.removeFromBus(..) : void
org.gjt.sp.jedit.EditBus.send(..) : void

visualizing results, encoding properties



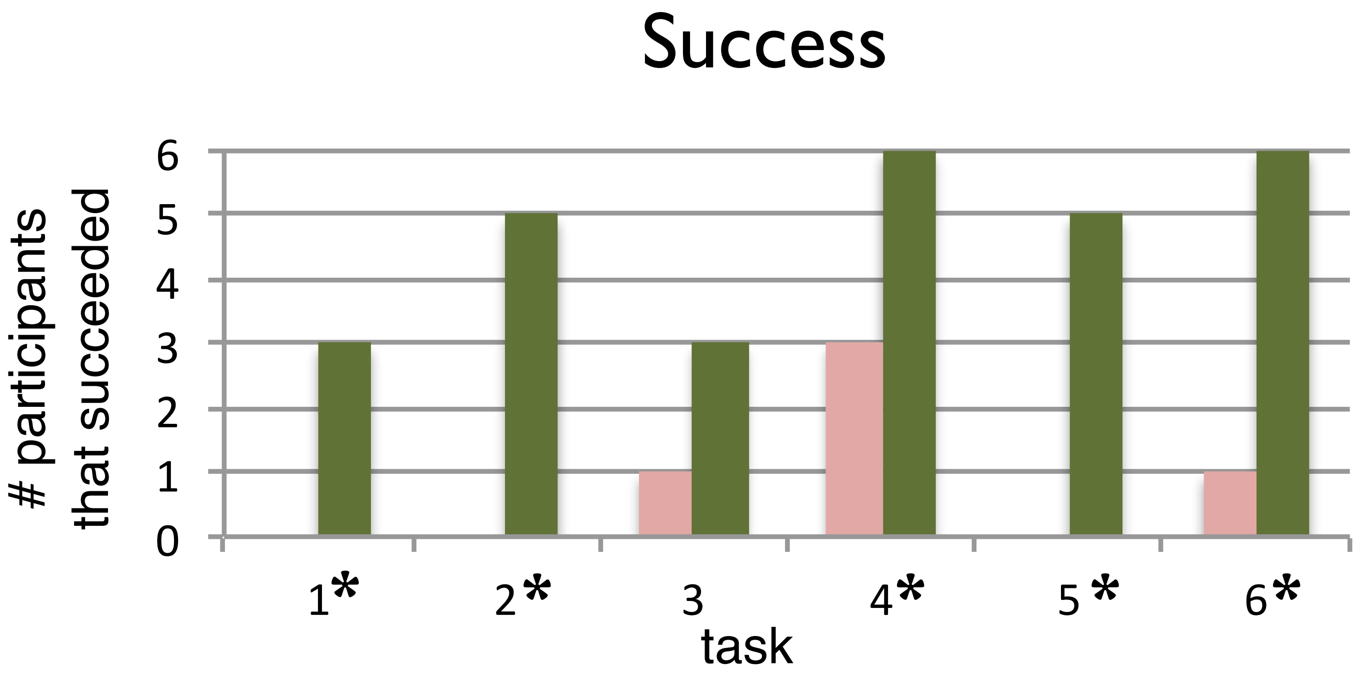
navigating IDE

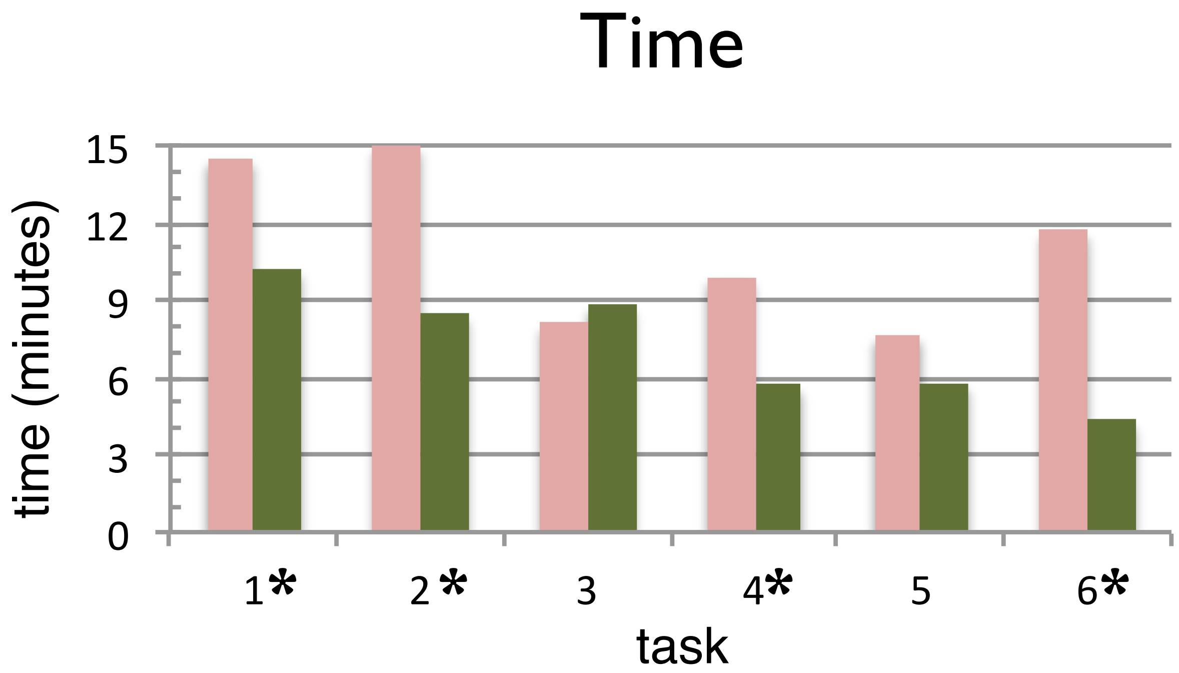


Results

Developers with Reacher **5.6** times more **successful** than working with Eclipse only.

Participants with Reacher took an average of **7.2** minutes vs. 11.1 minutes with Eclipse only (difference limited by ceiling effect).





* **significant** differences (p < .05)

pink **Eclipse** only
green Eclipse with **Reacher**

Control group traversed paths

- **Traversed** paths through code looking for targets
  - Relied heavily on **scent** - perceived relevance of method on path
  - E.g., to find EditBus messages, looked for important actions
  - Traversing through event listeners forced new search, often lost place
- Sometimes did **bidirectional** search
  - Started at origin and hypothesized destination
  - Tried to find connecting paths
- **Dynamic** investigation was difficult
  - Ran the program, but conditionals guarded path of interest
  - Did **static** investigation to figure out how to dynamically execute
  - But then was hard to determine which of many breakpoints hit it

Questions for Discussion

- Would you use this tool?
- In what contexts might Reacher be difficult to apply?
    - How might Reacher be extended?
- What are the pros and cons of static analysis vs. dynamic for debugging?
- What challenges might there be in commercializing Reacher?